Пермский филиал федерального государственного автономного
образовательного учреждения высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет экономики, менеджмента и бизнес-информатики

Виноградов Никита Андреевич

ОРГАНИЗАЦИЯ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ. ПОРОЖДАЮЩИЕ ПАТТЕРНЫ АБСТРАКТНАЯ ФАБРИКА И ОДИНОЧКА

Лабораторная работа

студента образовательной программы «Программная инженерия»
по направлению подготовки *09.03.04 Программная инженерия*

Руководитель
к.т.н., доцент кафедры Информационных технологий в бизнесе НИУ ВШЭ-Пермь

А.В. Кычкин

Пермь, 2020 год

Оглавление

Глава 1. Абстрактная фабрика

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

1.1. Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

1.2. Структура

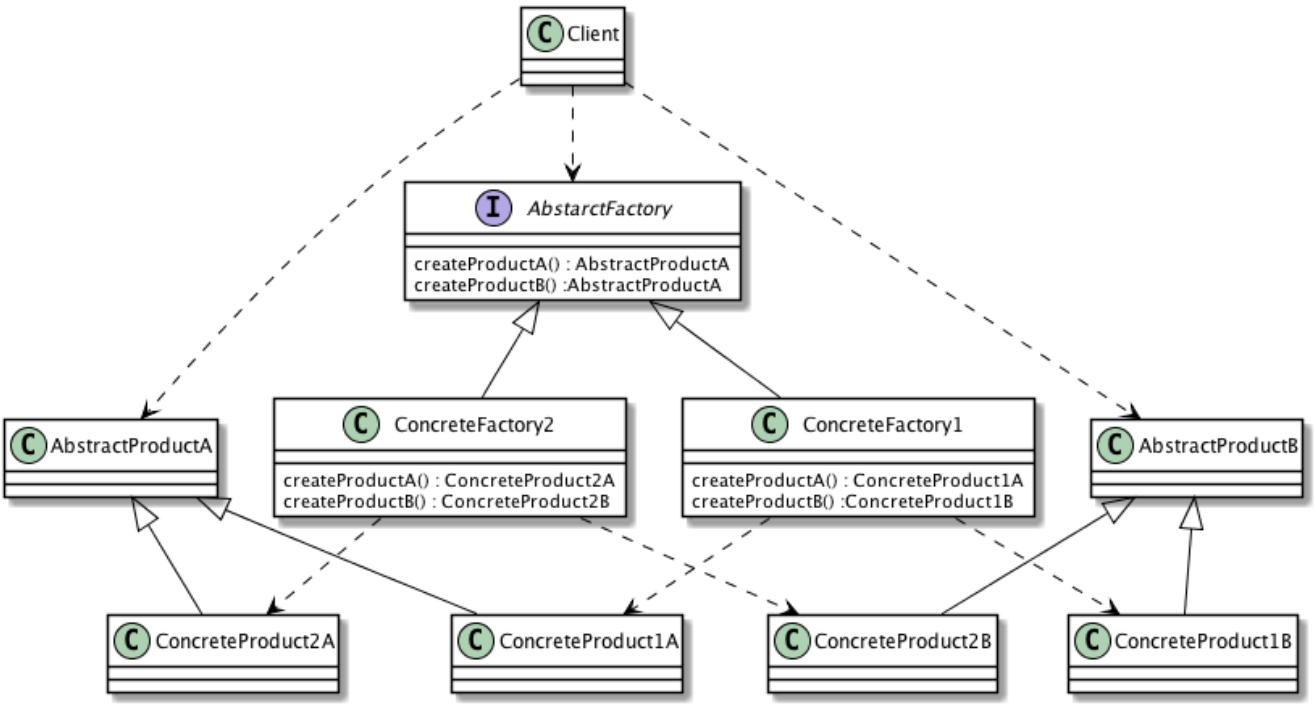

Рис. 1.1. Структура паттерна абстрактная фабрика

Участники:

- *AbstractFactory* - интерфейс предоставляющий действия для других фабрик
- *ConcreteFactory* - классы (ConcreteFactory1,ConcreteFactory2) выполняющий создание продуктов A1 и B1

- *AbstractProduct* - абстрактный объект (AbstractProductA,AbstractProductB) реализующий объект
- *ConcreteProduct* - классы (ConcreteProduct1A,ConcreteProduct1B,ConcreteProd реализующие классы объекты
- *Client* - класс пользующийся интерфейсами,которые объявлены в AbstractFactory и AbstractProduct

1.3. Способ применения

Данный паттерн применим для систем сборки проектов под разные платформы, также его можно использовать для подключений к базе данных.

Глава 2. Одиночка

Одиночка — это порождающий паттерн проектирования.

2.1. Назначение

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

2.2. Структура

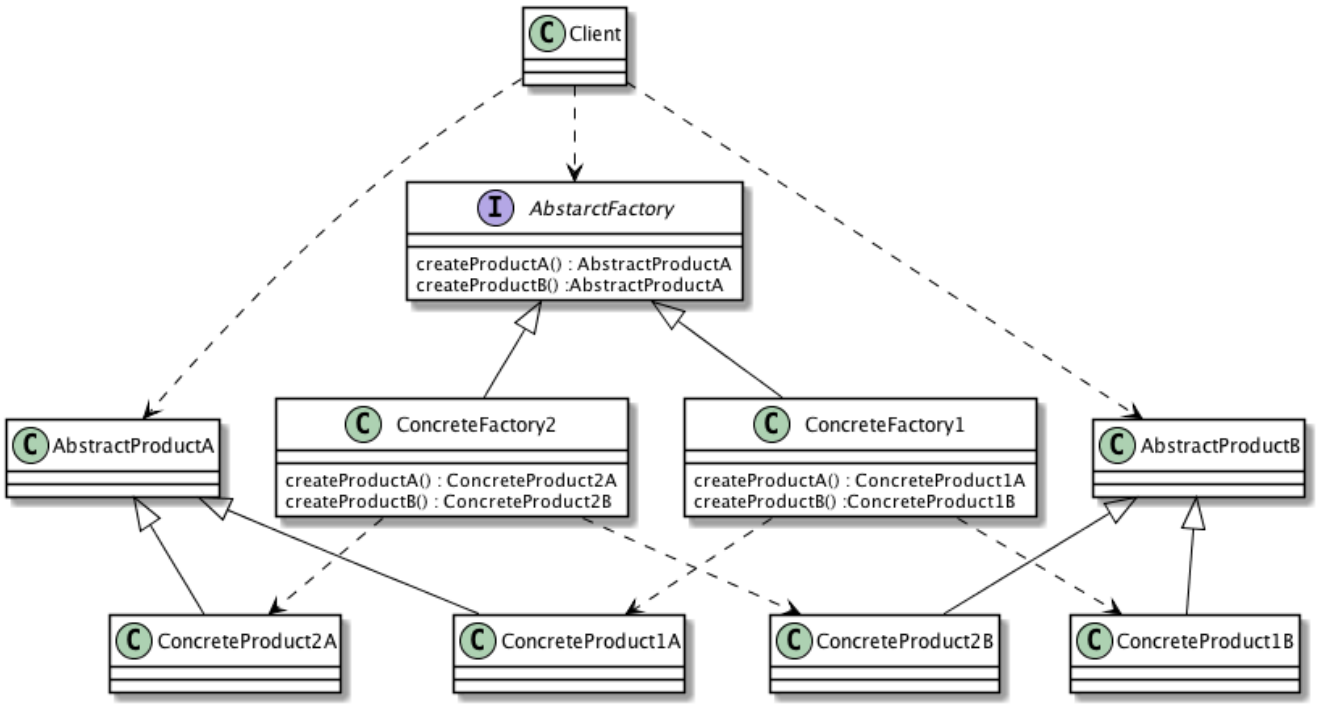

Рис. 2.1. Структура паттерна абстрактная фабрика

2.3. Способ применения

Данный паттерн применим когда в приложении есть глобальное состояние и данное состояние находится в одинственном экземпляре

Глава 3. Реализация паттернов

3.1. Диаграмма классов

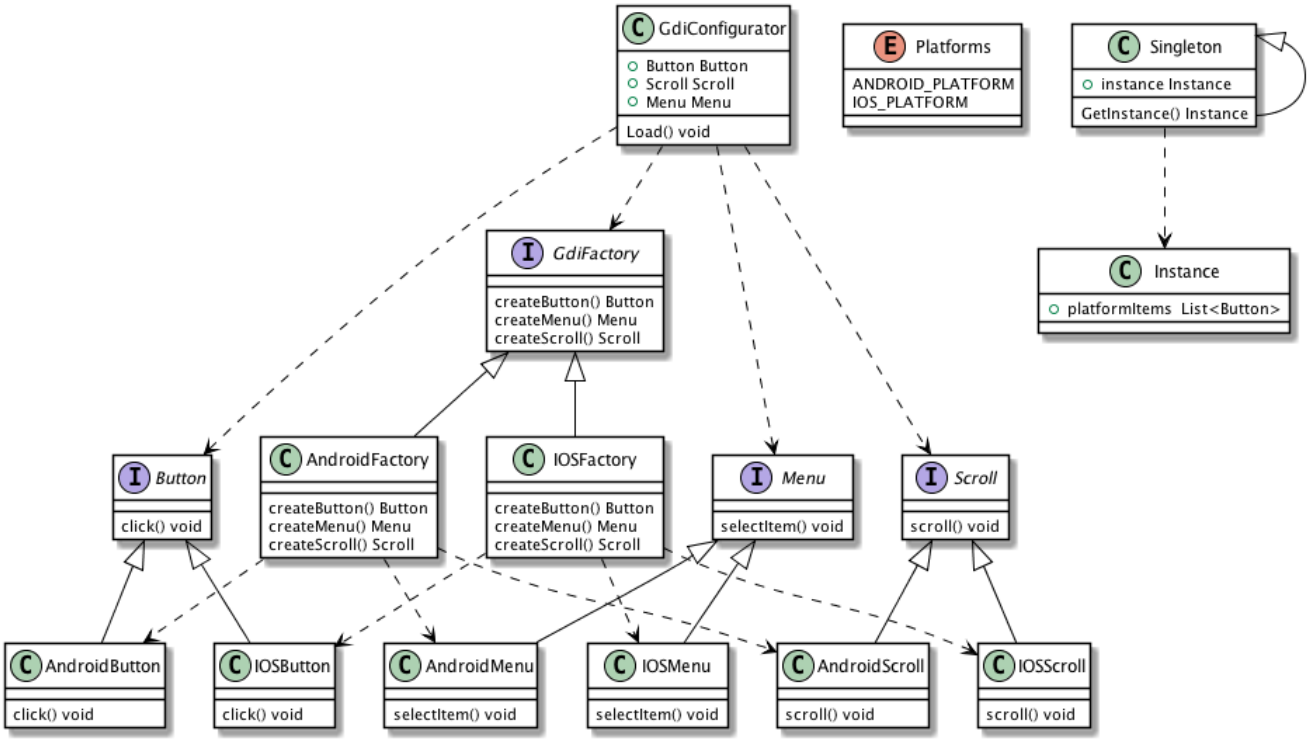

Рис. 3.1. Диаграмма классов

Участники:

- *GdiConfigurator* - класс имеющий в себе продукты фабрики и использующий по назначению
- *GdiFactory* - интерфейс определяющий стандартизированное поведение для фабрик
- *IOSFactory* - конкретная фабрика для создания объектов (IOSButton,IOSScroll,IOSMenu)
- *AndroidFactory* - конкретная фабрика для создания объектов (AndroidButton,AndroidScroll,AndroidMenu)
- *Scroll,Menu,Button* - интерфейсы определяющие стандартизированное поведение объектов

3.2. Диаграмма последовательности

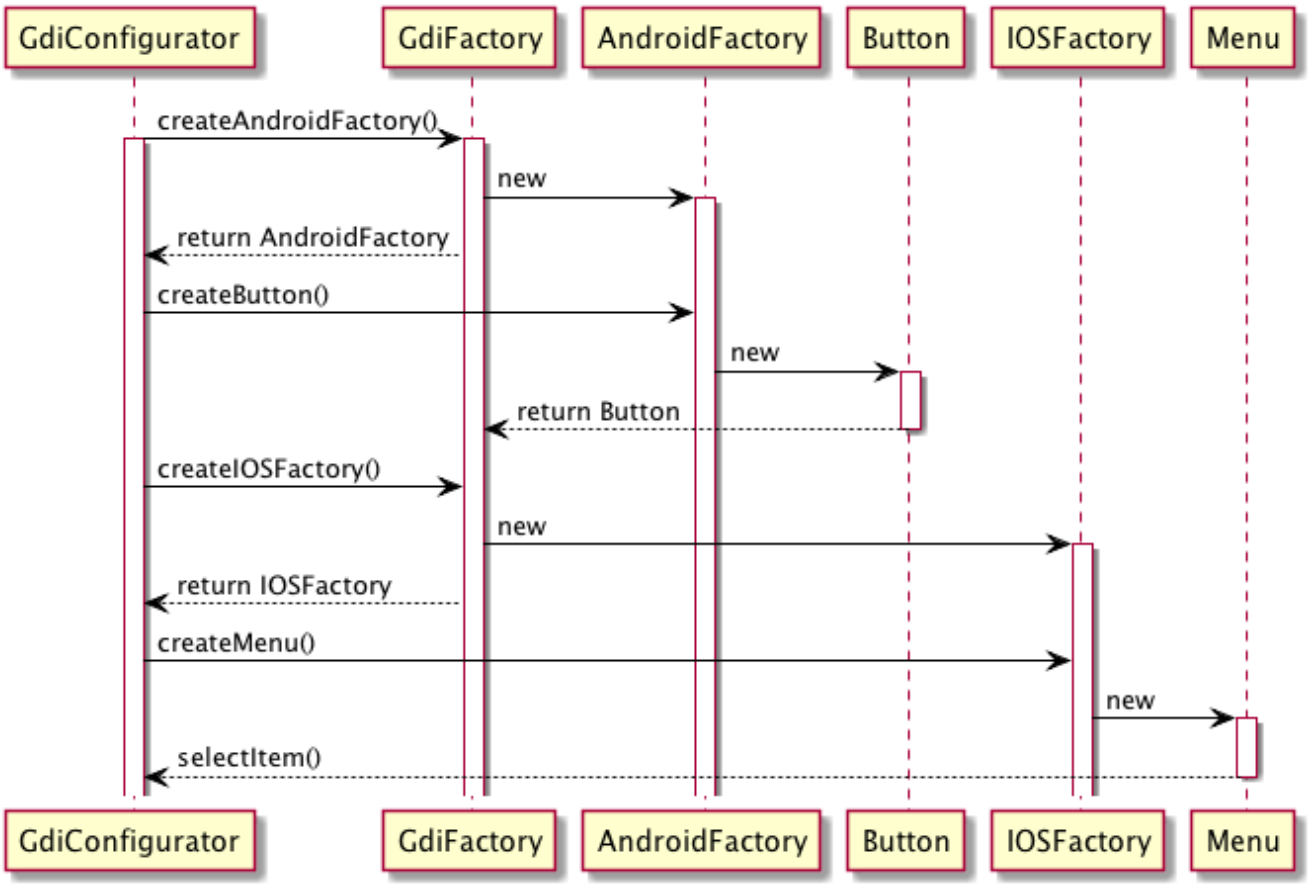

Рис. 3.2. Диаграмма последовательности

3.3. Код программы

```
package main

import (
"container/list"
"fmt"
)

const (
ANDROID_PLATFORM = 1
IOS_PLATFORM = 2
)
```

```
type (
Button interface {
click()
}
Scroll interface {
scroll()
}
Menu interface {
selectItem()
}
GdiFactory interface {
createButton() *Button
createMenu() *Menu
createScroll() *Scroll
}
)

/*
-----------------------------------------------------------------------------
IOS IMP
*/
type IOSFactory struct {
}

func NewIOSFactory() GdiFactory {
var g GdiFactory
g = IOSFactory{}
return g
}

func (I IOSFactory) createButton() *Button {
var b Button = IOSButton{}
return &b
}
```

```
func (I IOSFactory) createMenu() *Menu {
var m Menu = IOSMenu{}
return &m
}

func (I IOSFactory) createScroll() *Scroll {
var s Scroll = IOSScroll{}
return &s
}

type (
IOSButton struct {
}
IOSMenu struct {
}
IOSScroll struct {
}
)

func (I IOSButton) click() {
fmt.Println("click from ios button")
}
func (I IOSScroll) scroll() {
fmt.Println("scroll from ios scroll")
}
func (I IOSMenu) selectItem() {
fmt.Println("selectItem from ios menu")
}

/*
-----------------------------------------------------------------------

Android IMP
*/
```

```
type AndroidFactory struct {
}

func (I AndroidFactory) CreateButton() *Button {
panic("implement me")
}

func NewAndroidFactory() GdiFactory {
var g GdiFactory
g = AndroidFactory{}
return g
}
func (I AndroidFactory) createButton() *Button {
var b Button = AndroidButton{}
return &b
}

func (I AndroidFactory) createMenu() *Menu {
var m Menu = AndroidMenu{}
return &m
}

func (I AndroidFactory) createScroll() *Scroll {
var s Scroll = AndroidScroll{}
return &s
}

type (
AndroidButton struct {
}
AndroidMenu struct {
}
AndroidScroll struct {
}
```

```
)

func (I AndroidButton) click() {
fmt.Println("click from android button")
}
func (I AndroidScroll) scroll() {
fmt.Println("scroll from android scroll")
}
func (I AndroidMenu) selectItem() {
fmt.Println("selectItem from android menu")
}

/*
---------------------------------------------------------------------
GDI Configurator
*/
type GdiConfigurator struct {
Button Button
Menu Menu
Scroll Scroll
}

func NewGdiConfigurator(platform int) *GdiConfigurator {
var f GdiFactory
switch platform {
case ANDROID_PLATFORM:
f = NewAndroidFactory()
case IOS_PLATFORM:
f = NewIOSFactory()
}
return &GdiConfigurator{
Button: *f.createButton(),
Menu: *f.createMenu(),
Scroll: *f.createScroll(),
```

```
}

}
func Load(conf *GdiConfigurator) {
conf.Scroll.scroll()
conf.Button.click()
conf.Menu.selectItem()
}

/*
Signleton
*/
type singleton struct {
platformItems list.List
}

var instance *singleton

func GetInstance() *singleton {
if instance == nil {
instance = &singleton{} // ЭтоНЕпотокобезопасно -
instance.platformItems.Init()
}
return instance
}
func printList(items list.List) {
for e := items.Front(); e != nil; e = e.Next() {
fmt.Println("%#v", e.Value)
}
}
func main() {
c := NewGdiConfigurator(IOS_PLATFORM)
Load(c)
GetInstance().platformItems.PushBack(c.Menu)
```

```
printList(GetInstance().platformItems)
}
```